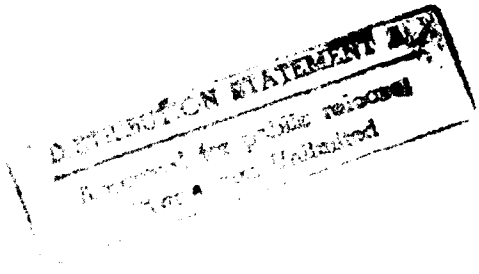
Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors

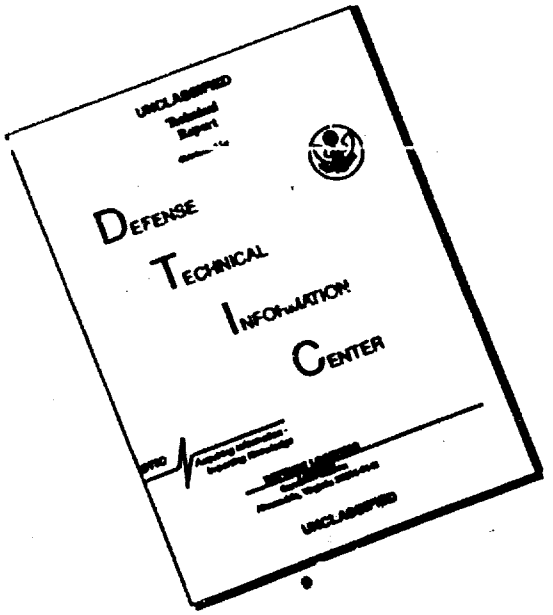
Ricardo Bianchini, Mark E. Crovella,
Leonidas Kontothanassis and Thomas J. LeBlanc

Technical Report 449
April 1993



UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors

Ricardo Bianchini, Mark E. Crovella,
Leonidas Kontothanassis and Thomas J. LeBlanc

{ricardo,crovella,kthanasi,leblanc}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 449

April 1993

Abstract

Memory contention can be a major source of overhead in large-scale shared-memory multiprocessors. Although there are many hardware solutions to the problem of memory contention, these solutions are often complex and expensive, so software solutions are an attractive alternative. This paper evaluates one particular software solution, called *block-column allocation*, which is very effective at reducing memory contention for a large class of SPMD (Single-Program-Multiple-Data) programs, and can be implemented easily by the compiler. We first quantify the impact of memory contention on performance by simulating the execution of several application kernels on a large-scale multiprocessor. Our simulation results confirm that memory contention is widespread on large-scale machines; our applications suggest that contention is usually caused by synchronized access to a range of addresses (rather than to a single address). We show that block-column allocation, where each range of addresses is divided into cache lines, and each cache line is allocated to a separate memory module, can nearly eliminate this source of memory contention. As our main contribution, we compare block-column allocation to row-major allocation (a common data allocation scheme) and logarithmic broadcasting (the standard software technique for alleviating memory contention). Our analysis demonstrates the clear superiority of block-column allocation over row-major allocation in the presence of memory contention. Our analysis also indicates that the choice between block-column allocation and logarithmic broadcasting is less clear, as it depends both on the type of synchronization used and the number of processors. We can conclude however that on large-scale machines with hundreds of processors, block-column allocation and lock-based synchronization is the most effective combination for reducing memory contention in SPMD matrix computations.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930). Ricardo Bianchini is supported by Brazilian CAPES and NUTES/UFRJ fellowships. Mark Crovella is partially supported by a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1993	technical report

4. TITLE AND SUBTITLE

Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors

5. FUNDING NUMBERS

ONR N00014-92-J-1801

6. AUTHOR(S)

Ricardo Bianchini, Mark E. Crovella, Leonidas Kontothanassis, and Thomas J. LeBlanc

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
734 Computer Studies Bldg.
University of Rochester
Rochester, NY 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER

TR 449

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, VA 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Memory contention can be a major source of overhead in large-scale shared-memory multiprocessors. There are many hardware solutions, but they are often complex and expensive, so software solutions are an attractive alternative. This paper evaluates one solution, block-column allocation, which is very effective at reducing memory contention for a large class of SPMD (Single-Program-Multiple-Data) programs, and can be implemented easily by the compiler. We first quantify the impact of memory contention on performance by simulating the execution of several application kernels on a large-scale multiprocessor. Our simulation results confirm that memory contention is widespread on large-scale machines; our applications suggest that contention is usually caused by synchronized access to a range of addresses (rather than to a single address). We show that block-column allocation can nearly eliminate this source of memory contention. As our main contribution, we compare block-column allocation to row-major allocation and logarithmic broadcasting. Our analysis demonstrates the clear superiority of block-column allocation over row-major allocation in the presence of memory contention.

14. SUBJECT TERMS

memory contention; matrix computations; broadcasting; block-column allocation

15. NUMBER OF PAGES

21

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

1 Introduction

Large-scale shared-memory multiprocessors based on off-the-shelf RISC microprocessors offer the computational power needed to solve large scientific problems. Several efforts are underway to design shared-memory multiprocessors that scale to hundreds of processors, including the Stanford DASH [Lenoski *et al.*, 1992] and MIT Alewife [Agarwal *et al.*, 1992] machines. To effectively exploit these large-scale multiprocessors we must eliminate *all* bottlenecks that limit scalability. One such bottleneck is memory contention, which occurs whenever multiple processors need to access the same memory module simultaneously. Absent special hardware, the memory module can satisfy only one request at a time, and must reject all others. By serializing multiple accesses, and requiring that a processor repeatedly issue requests to the same memory module so as to satisfy a single data reference, memory contention can severely limit the speedup of parallel applications.

The majority of techniques for alleviating memory contention assume special hardware support, such as multi-stage interconnection networks with combining of memory references [Gottlieb *et al.*, 1983; Pfister and Norton, 1985], interleaved memory [Pfister *et al.*, 1985], and eager sharing [Wittie and Maples, 1989]. Although these techniques are known to reduce or eliminate memory contention, the associated hardware can be both complex and expensive, and may depend on particular properties of the interconnection network.

One general technique to alleviate memory contention caused by simultaneous access to data is to broadcast the data to every processor [Saad and Schultz, 1989]. This technique is especially easy to implement on machines that support broadcasting in hardware, such as bus-based multiprocessors like the Sequent Symmetry, or ring-based systems like the Kendall Square KSR1. On direct-connected distributed-memory machines, such as DASH and Alewife, the broadcast must be implemented in software (or by the coherency protocol). Software broadcasting typically employs a logarithmic distribution of data, wherein the source processor sends the data to N other processors, each of which pass the data on to N other processors, until all processors receive the data.

Two other related techniques for alleviating contention are software combining trees [Yew *et al.*, 1987] and data replication. Software combining trees are analogous to hardware combining networks, and incorporate logarithmic broadcasting. As in combining networks, requests for shared data flow from the leaves of the tree to the root, and the data flows from the root down to the leaves. As in logarithmic broadcasting, the data flows down the tree in $O(\log P)$ steps (where P is the number of processors), and the width of the tree limits the potential for memory contention. In SPMD programs where all processors are known to require the data, there is no need to propagate read requests up the tree, and therefore no need to distinguish software combining trees and logarithmic broadcasting.

We can also limit memory contention by replicating data across multiple memory modules. By distributing the requests for data evenly among the copies, we can reduce or eliminate memory contention for the original copy. Although this scheme avoids broadcasting data to all the processors (since each processor explicitly requests the data from one of the copies), it requires that the copies either be created serially, or that some form of broadcasting be used to create the copies. In addition, there can still be memory contention for an individual copy, depending on the number of processors and the number of copies. See [Bianchini *et al.*, 1993] for a complete discussion of this technique.

Each of the techniques described above is general enough to use in any program. However,

our investigation of memory contention in programs for solving linear algebra and graph problems suggests that techniques devoted specifically to parallel matrix computations [Geist *et al.*, 1987; Ortega and Romine, 1988] can also be very effective at alleviating contention. In this paper, we focus on one such technique, called *block-column allocation*. This technique is motivated by the observation that memory contention in matrix computations is typically caused by simultaneous access to a single row of the matrix by multiple processors. If matrices are allocated among memories by rows, simultaneous access to any part of a row requires that processors contend for a single memory module. Allocating matrices among memories by column alleviates this source of contention, but creates other problems, such as false sharing. In block-column allocation, a row is divided into cache lines, and the cache lines are distributed among the memories in round-robin order. This technique has the spatial locality properties of allocation by rows, and the memory contention properties of allocation by column.

Using block-column allocation to alleviate memory contention is not new; the same basic idea (called *interleaved shared memory*) is supported in hardware on the BBN TC2000 [BBN, 1989]. Although this hardware feature has been used in scientific applications [Amestoy *et al.*, 1992; Brooks and Warren, 1991], we know of no comprehensive evaluation of this technique, or of any published experiences with this technique when applied in software. We seek to characterize the source and extent of memory contention in SPMD matrix computations, quantify the costs and benefits of block-column allocation, and evaluate the tradeoffs between block-column allocation and logarithmic broadcasting on large direct-connected shared-memory multiprocessors.

In the following section we describe our example application programs, and use simulation to quantify the impact of memory contention on their performance. In section 3 we describe *implementations* of our example programs based on block-column allocation, and quantify the effect of our implementation on the latency of remote memory accesses and the running time of our applications. Our most important contributions are presented in section 4, where we analyze the costs and benefits of block-column allocation, and compare its performance to both row-major allocation and logarithmic broadcasting. We present our conclusions in section 5.

2 Characterizing the Effects and Source of Memory Contention

Memory contention occurs whenever multiple processors require simultaneous access to a single memory module, thereby producing a so-called *hot spot*. Glenn *et al.* [1991] divided hot spots into three categories: 1) read-only memory with a large number of readers; 2) synchronized access to memory modules due to related strides; and 3) hot spots caused by synchronization references. Type 3 hot spots can be effectively eliminated using proper synchronization techniques [Mellor-Crummey and Scott, 1991]. Type 2 hot spots are present in certain highly structured problems, and are relatively uncommon. Here we consider four representative SPMD programs that exhibit a form of type 1 memory contention.

Our example programs all require that all processors simultaneously access data that was recently modified by a single processor. This form of producer/consumers relationship can lead to memory contention if the data that must be accessed by all processors resides in a single memory module. As we will show, the resulting memory contention can significantly degrade performance.

2.1 Applications

Our example programs are drawn from two large classes of applications: linear algebra and graph algorithms. These SPMD programs represent computational kernels similar to those found in many applications. For each kernel, matrix data is allocated in row-major order.

The first program is a parallel implementation of Gaussian elimination (without pivoting or back-substitution). The code for this program is as follows:

```
FOR pivot = 1 TO N-1 DO
    FORALL row = pivot+1 TO N DO
        tmp = M[row][pivot]/M[pivot][pivot]
        FOR col = pivot TO N DO
            M[row][col] = M[row][col] - M[pivot][col] * tmp
```

On each iteration of the outermost sequential loop, we create a set of processes, each of which eliminates the entries in a single row of the input matrix. All processes require access to the same pivot row, and since all processes begin execution at approximately the same time, the pivot row is a likely source of memory contention. In our experiments we used a random matrix of size 512 $\times$ 512 as input.

Our second program implements matrix inversion. The code for this program is:

```
FOR pivot = 1 TO N-1 DO
    FORALL row = pivot+1 TO N DO
        M[row][pivot] = M[row][pivot]/M[pivot][pivot]
        tmp = M[row][pivot]
        FOR col = pivot+1 TO N DO
            M[row][col] = M[row][col] - M[pivot][col] * tmp

FORALL row = 1 TO N DO
    FOR diagonal = 1 TO N DO
        sum = 0
        FOR col = 1 TO diagonal DO
            sum = sum + M[diagonal][col] * I[col]
        I[diagonal] = I[diagonal] - sum
    FOR diagonal = N DOWNTO 1 DO
        sum = 0
        FOR col = N DOWNTO diagonal DO
            sum = sum + M[diagonal][col] * M[row][col]
        MInv[row][diagonal] = (I[diagonal] - sum) / M[diagonal][diagonal]
```

The first phase of this program uses L-U decomposition, which has roughly the same structure as Gaussian elimination, and therefore is susceptible to memory contention. Although both sequential loops within the second phase require that all processes access the same row of matrix M, the second of these loops cannot cause contention; M has already been loaded into the local cache during execution of L-U decomposition and the first loop. The first sequential loop in the second phase may suffer from contention, but the processes are only loosely synchronized, and are likely to skew their accesses to the matrix during execution. (There is no contention for accesses to I, since each

process has its own version of this data structure.) Again, we used a random input matrix of size 512×512.

Our third program computes the transitive closure of a graph, which is represented using an adjacency matrix stored in row-major order. The code for this program is as follows:

```
FOR i = 1 TO N DO
    FORALL j = 1 TO N DO
        IF M[j][i] THEN
            FOR k = 1 TO N DO
                IF M[i][k] THEN M[j][k] = TRUE
```

Our sample input graph has 512 vertices, and each vertex is connected to each other vertex with probability 0.5. Unlike the previous two programs, where each process does roughly the same amount of work, there is the potential for load imbalance in this program. Some processes do $O(N)$ work, while others do $O(1)$ work. Each process that does $O(N)$ work must access the same row, represented by the index of the outermost loop. As with the previous programs, access to this row may introduce memory contention, but we would expect the effects of contention to be less in this case since not all processes execute the innermost loop.

Our final program uses a parallelization of the Warshall-Floyd algorithm to compute the all-pairs shortest paths for a graph with 400 vertices. The code for this program is as follows:

```
FOR k = 1 TO N DO
    FORALL i = 1 TO N DO
        IF G[i][k] < INFINITY THEN
            FOR j = 1 TO N DO
                IF G[i][k] + G[k][j] < G[i][j] THEN
                    G[i][j] = G[i][k] + G[k][j]
```

As in the previous examples, all processes require access to the same row of the input matrix. This program differs from the previous example in that the matrix elements represent distances between vertices rather than a boolean value of connectivity. Since we use four-byte integers to represent distances in the all-pairs shortest paths program, and single bytes to represent connectivity in transitive closure, there is more communication required in the all-pairs shortest paths program, even though both programs do roughly the same amount of work. We expect therefore that memory contention will have a greater impact on the all-pairs shortest paths program.

We selected these computational kernels to illustrate the tradeoffs that must be considered in the face of memory contention. Gaussian elimination is both common in practice, and illustrative of one common source of memory contention. Matrix inversion can benefit from any technique used to alleviate memory contention during Gaussian elimination (since the same technique can be applied to L-U decomposition), but the second phase of matrix inversion may incur overhead due to changes in the data allocation scheme used to alleviate contention. Also, matrix inversion illustrates simultaneous accesses that cause contention (during L-U decomposition), and simultaneous accesses that do not cause contention (during the second phase, when the matrix is already loaded into each local cache).

Since both barriers and locks (used as condition variables) can be used to implement the necessary producer/consumer synchronization in these problems, we also explore the role of synchronization. We implemented the parallel loop in L-U decomposition using a barrier, which synchronizes

all processes on each iteration of the outermost loop, and thereby increases the potential for memory contention during access to the pivot row. We used locks (as a form of condition variable) to implement synchronization in Gaussian elimination, which allows each process a bit more freedom during execution, and thereby reduces contention.

Transitive closure (implemented with locks) is interesting because it is similar in structure to Gaussian elimination, except that conditional execution of the inner loop helps to alleviate contention. All-pairs differs from transitive closure in that (a) we used a barrier to implement synchronization, and (b) each element of the matrix is a 4-byte distance, rather than a single byte representing connectivity.

2.2 Methodology

Since we are interested in studying memory contention in the truly large-scale shared-memory multiprocessors currently under development, direct experimentation is not an available option. Thus, we use analytic modeling and simulation for our studies. We simulate a large-scale direct-connected multiprocessor (up to 256 processors) executing our example applications. Our simulations consist of two distinct steps: a trace collection process, and a trace analysis process. The trace-collection step uses Tango [Davis *et al.*, 1991] to simulate a multiprocessor with (infinite) coherent caches. The traces generated by Tango contain the data references that missed in the local cache of each processor, and all synchronization events.

Our analyzer process takes as input an address trace produced by Tango, and simulates execution of the references in the trace on a distributed shared memory multiprocessor. The analyzer assigns each reference to the appropriate processor at the appropriate time by tracking the delay induced by previous references, combined with the time spent executing instructions on the processor. The analyzer respects the synchronization behavior of an application as represented by the synchronization events contained in the trace. Synchronization events are not allowed to cause contention in our model, although they are critical in maintaining the relative timing of events during trace analysis.

In our machine model, a memory module can process only one request at a time. Requests arriving when the module is busy are rejected and must be reissued. Our analyzer measures contention for memory at the page level; thus each 4KB page is treated as a separate memory module to which requests may be directed. We treat each page as a separate memory module so as to simulate an ideal page placement policy in which contention caused by simultaneous accesses to multiple pages does not occur. One consequence of this assumption is that the number of memory modules in the system is dependent on the size of the problem and not on the number of processors in use. As a result, our estimates of memory contention are optimistic, in that we measure the contention inherent in an application, independent of the placement of pages in memory modules.

Our simulations assume a cache line size of 64 bytes, a fixed network latency of 36 processor cycles, and local memory latency of 10 processor cycles. In the absence of contention, a remote memory request requires a request message, a reply message, and memory service time, or 82 cycles total. Each request rejected due to contention suffers a 72 cycle penalty, corresponding to an immediate re-issue of the request.

Our simulation assumptions are optimistic, in that we chose values for the simulation parameters that are likely to result in less contention than would exist in the machines of the near future, and

yet still produce substantial contention in our simulations. For example, we chose to use 64-byte cache lines, which are larger than the cache lines used in both DASH and Alewife (but smaller than the cache lines used in the Kendall Square KSR1). Given the spatial locality in our applications and the lack of fine-grain sharing, we would expect these longer lines to result in fewer cache misses, and less memory contention than would occur in either DASH or Alewife. Similarly, we chose a memory latency of 10 processor cycles per cache line, and a network latency of 36 processor cycles, both of which are quite optimistic. We would expect the faster remote memory service time represented by these two factors to result in less memory contention than would occur in DASH and Alewife. Our infinite cache assumption means that we only measure the effect of invalidation-related misses, and ignore capacity misses. Our assumption that network latency is fixed (i.e., there is no network contention) allows us to isolate the effects of memory contention from network contention; adding network contention to our simulations would assign some of the contention we observe to the network rather than the memory, but would not be likely to affect the tradeoffs we consider here.

2.3 The Effects of Memory Contention

We simulated each of our application programs, and measured the number of remote memory accesses, the number of remote memory accesses delayed by memory contention, the average latency of remote access, and the running time. The results are shown in Tables 1-3.

As seen in Table 1, memory contention is widespread in our applications. On a 200 processor machine, over 50% of all remote memory accesses are delayed due to memory contention. Even on 50 processors, 46% of all remote references are delayed during matrix inversion, and 89% of all remote references are delayed in the all-pairs shortest paths program. Although only 8% of all remote accesses are delayed during transitive closure on 50 processors, the percentage rises to 60% on 200 processors. There is a similar rise in the percentage of references that experience contention in Gaussian elimination. Only matrix inversion exhibits a consistent degree of contention, with roughly 50% of all remote references experiencing delays on 50, 100, and 200 processors.

The minimum delay introduced by memory contention is 72 cycles of network round-trip latency, but much greater delay is possible, since subsequent requests may also be rejected due to contention. Table 2 illustrates the effect of memory contention on the effective latency of remote memory accesses. The minimum possible latency is 82 cycles, which represents the round-trip network costs, and the latency of the memory. As seen in Table 2, transitive closure suffers a 16% slowdown in remote memory access time due to contention on 50 processors, while every other program suffers at least a 100% slowdown. The all-pairs program suffers the worst: a 3270% slowdown in the average latency of remote accesses due to memory contention! As we increase the number of processors, the latency of remote memory accesses rises dramatically in every case. Even transitive closure, which has the least contention, suffers a slowdown of remote memory accesses of 655% on 200 processors. These results suggest that memory contention will be a serious problem on large-scale machines, and yet all of these results are optimistic, since each 4KB page is considered a separate memory module in our simulations.

The effect of memory contention on application performance isn't obvious from these tables, since it depends on the frequency of remote references. Table 3 shows how memory contention affects the running time of our applications. For Gaussian elimination and all-pairs shortest paths,

Application	Percent of Delayed Misses		
	50 processors	100 processors	200 processors
Gaussian elimination	20%	56%	84%
Matrix inversion	46%	51%	51%
Transitive closure	8%	35%	60%
All pairs	89%	92%	94%

Table 1: Percent of all remote memory references that experience delay due to memory contention under row-major allocation.

Application	Average Remote Memory Latency		
	50 processors	100 processors	200 processors
Gaussian elimination	164	572	1546
Matrix inversion	264	536	991
Transitive closure	95	228	619
All pairs	2764	5924	12335

Table 2: Effect of memory contention on average latency of remote memory accesses (in cycles) under row-major allocation.

memory contention causes the running time to increase with an increase in processors. In fact, moving from 50 to 200 processors increases the running time of these applications by a factor of 2-3, rather than cutting the running time by a factor of 4. The situation is not quite as bleak in the case of matrix inversion, where 100 processors perform slightly better than 50 processors; however, 200 processors perform no better than 50 processors. Transitive closure is the only program that benefits from an increase in processors, although doubling the number of processors from 50 to 100 only improves performance by a factor of 1.8, and multiplying the number of processors by 4 only improves performance by a factor of 2.4. It is important to note that, for the inputs used in our simulations, these programs have good locality of reference and load balancing properties, and achieve good speedup when contention is not considered. Thus, for all of these programs, memory

Application	Running Time		
	50 processors	100 processors	200 processors
Gaussian elimination	7.4	8.5	15.6
Matrix inversion	26.1	21.7	26.0
Transitive closure	21.7	12.3	9.0
All pairs	43.0	71.3	136.8

Table 3: Effect of memory contention on running time (in millions of cycles) under row-major allocation.

contention is the major obstacle to effective speedup.

The effects of contention are magnified even more if we relax some of our optimistic assumptions. For example, if we double the memory latency to 20 processor cycles, the effect of contention is even more pronounced. On 200 processors, 92% of the misses in Gaussian elimination suffer contention (up from 84%), the average remote reference latency increases to 2910 cycles (up from 1546), and the running time increases to 28.8 M cycles (up from 15.6 M cycles). Similarly, if we keep memory latency at 10 cycles and reduce the cache line size to 32 bytes, then 90% of the misses in Gaussian elimination suffer contention, the average remote latency increases slightly to 1571 cycles, and the running time increases dramatically to 30.9 M cycles (since we've doubled the number of remote references). If we both double the memory latency and reduce the cache line size to 32 bytes, then the average remote latency increases to 2904 cycles, and the running time increases to 55.2 M cycles. These results suggest that under less optimistic (and perhaps more realistic) assumptions, memory contention is likely to be an extremely serious problem in large-scale shared-memory machines.

2.4 The Source of Memory Contention

From the results presented in the previous section, it is obvious that all of our example programs suffer from memory contention. This contention could be caused by any of three factors: (1) simultaneous access to a single element of the matrix, (2) simultaneous access to a single row of the matrix (which resides in a single page, and therefore results in memory contention), and (3) simultaneous access to multiple rows that happen to reside in the same page. In all of our examples, we padded the rows of the matrix to fill a page, and therefore eliminated any contention caused by *simultaneous access to multiple rows within a single page*. Simultaneous access to a single element of the matrix *can* occur in our programs since, upon creation, all processes immediately try to reference the first element of a row in the matrix. However, serial access to the first element in a row tends to skew the requests for subsequent elements in that row, thereby avoiding contention for individual elements.

We validated this hypothesis by a simple experiment in which we simulated Gaussian elimination on 50 processors, using a matrix that was allocated so that elements within the same row were placed in different pages. This allocation strategy reduced the percentage of delayed references from 20% to 1.5%, and the average remote access latency from 164 cycles to 83 cycles. This experiment confirms that the memory contention seen in our examples is due primarily to simultaneous access to the elements of a row, all of which reside in one memory module.

We can also see from our examples that synchronization plays an important role in memory contention. All-pairs shortest paths experiences the worst contention by far, in part because our implementation uses barriers to implement the parallel loop. Transitive closure is similar in structure, but we used locks in its implementation. By using barriers in the all-pairs shortest paths program, we force all processes to access the same row at the same time on every iteration of the outermost loop, thereby increasing contention. To confirm the role of barrier synchronization as a root cause of memory contention in all-pairs shortest paths, we implemented the program using locks instead of barriers on 50 processors. The percentage of remote references that were delayed fell from 89% to 54% as a result of this change. More importantly, the average latency of a remote memory access fell from 2764 cycles to 247 cycles, and the running time decreased from 43M cycles

to 14.4M cycles. It is clear from this experiment that barriers exacerbate the problem of memory contention.[1]

The effect of barriers can also be seen in the performance of matrix inversion, which uses barriers in the implementation of the L-U decomposition step. On 50 processors, matrix inversion suffers enormous contention in the L-U decomposition step, where processes are tightly synchronized, but not in the following step, which has no synchronization. Gaussian elimination suffers contention throughout execution, but not as much as L-U decomposition, since we use locks in the implementation of Gaussian elimination. As we increase the number of processors, a greater percentage of remote references exhibit contention in Gaussian elimination (since all remote references are susceptible to contention), while contention is confined to references during the L-U decomposition step of matrix inversion (which only contains about half of all remote references in the application). Therefore, the percentage of delayed references continues to rise in Gaussian elimination as we increase the number of processors, but remains around 50% in matrix inversion.

We conclude from these experiments that the major source of contention in our application programs is due to synchronized access to the elements of a single row of the matrix, all of which reside in a single page (or memory module). Although relaxing synchronization constraints (by replacing barriers with locks) helps to reduce contention, we still observe substantial performance degradation due to contention in large-scale machines. In the next section we consider an alternative data allocation strategy designed to address this problem.

3 Reducing Memory Contention with Block-Column Allocation

Our experiments in the previous section suggest that the main cause of memory contention in our example programs is the row-major allocation we used for matrices. Row-major allocation places an entire row of the matrix in a single page (or memory module), so that access to the row by multiple processors results in memory contention. Since none of our example programs access a matrix by columns, one obvious way to alleviate memory contention is to allocate the matrices in column-major order. That way, each element of a row resides in a different memory module.

We simulated Gaussian elimination on 50 processors using column-major allocation. In this implementation, every element of a row resides in a different page. This implementation is successful at reducing memory contention; only 1.5% of all remote references experience a delay, and the average delay is only 83 cycles. However, this implementation also introduces 15 times as many cache misses (due to false sharing), and increases the running time from 7.4M cycles to 30.2M cycles! We can see from this experiment that column-major allocation merely trades memory contention for additional cache misses, and does not solve the performance problem. We require an allocation strategy that has the spatial locality properties of row-major allocation, and the memory contention properties of column-major allocation. *Block-column allocation* has both properties.

[1] Note that the effects of memory contention are greater in the lock-based implementation of all-pairs shortest paths than in the lock-based implementation of transitive closure, since there are many more cache misses in all-pairs shortest paths.

3.1 Block-Column Allocation

In block-column allocation, we divide each row of the input matrix into cache blocks, and map the cache blocks of a single row into different memory modules. In effect, we use column-major allocation of cache blocks, rather than column-major allocation of elements. Since no cache block contains elements from multiple rows, we eliminate the additional cache misses due to false sharing in column-major allocation. Since the cache blocks of a single row map to different memory modules, no memory contention occurs when multiple processors simultaneously access different cache blocks of the same row.

The algorithm changes needed to exploit block-column allocation can be described in terms of two loop transformations: strip-mining followed by loop interchange. We use strip-mining on the innermost loop to group together the elements of a row that fit within one cache block. We then interchange the innermost loop with the enclosing loop, so that we iterate over columns of cache blocks. These are standard loop transformations performed by compilers; block-column allocation requires that compilers accompany these transformations with corresponding changes in data allocation.

The performance benefits of block-column allocation can be seen in Tables 4-6. As seen in Table 4, the percentage of remote references that experience delay has dropped dramatically under block-column allocation.[2] In most cases, less than 2% of all remote references experience delay. Even in the worst case (all-pairs shortest paths on 200 processors), only 6.3% of all remote references experience delay. By way of contrast, 94% of all remote references experience delay when simulating all-pairs shortest paths on 200 processors using row-major allocation.

Table 5 shows the effect of block-column allocation on the average latency of remote memory accesses. For Gaussian elimination, the average remote access latency on 200 processors is 82 cycles, which is optimal. The results for transitive closure are also close to optimal. Average latency for matrix inversion under block-column allocation increases slightly with an increase in processors, but still manages a 6-10 fold decrease in average latency when compared with row-major allocation. And even though all-pairs shortest paths still suffers from contention, which results in an average remote access latency of 366 cycles on 200 processors, block-column allocation improves the average remote access latency by a factor of 18 to 33.

This decrease in remote access latency produces a corresponding improvement in running time, as seen in Table 6. Under block-column allocation, each of our applications runs faster with an increase in processors. For Gaussian elimination and transitive closure, doubling the number of processors cuts the running time nearly in half. Additional processors also improve the running time of matrix inversion, although not in the same proportion. Even all-pairs shortest paths continues to exhibit improved running time with an increase in processors, although the performance improvements offered by 200 processors are insignificant. The speedup of matrix inversion and all-pairs shortest path are both limited by the use of barrier synchronization; too many processors waste cycles waiting for a barrier.

Block-column allocation is also effective at reducing contention under less optimistic assumptions than those used in the majority of our experiments. For example, even if we double the memory latency to 20 cycles, block-column allocation eliminates most memory contention in Gaussian elimination. On 200 processors, only 0.78% of the misses suffer contention, the average latency

[2]The actual number of cache misses is the same for both block-column allocation and row-major allocation.

Application	Percent of Delayed Misses		
	50 processors	100 processors	200 processors
Gaussian elimination	0.16%	0.12%	0.27%
Matrix inversion	1.9%	2.1%	2.1%
Transitive closure	0.4%	0.9%	1.8%
All pairs	5.9%	6.2%	6.3%

Table 4: Percent of all remote memory references that experience delay due to memory contention under block-column allocation.

Application	Average Remote Memory Latency		
	50 processors	100 processors	200 processors
Gaussian elimination	82	82	82
Matrix inversion	87	92	99
Transitive closure	83	84	86
All pairs	150	222	366

Table 5: Effect of memory contention on average latency of remote memory accesses (in cycles) under block-column allocation.

of remote accesses is only 95 cycles, and the running time only increases by 15%. The same observation applies if we reduce the cache line size to 32 bytes. For Gaussian elimination on 200 processors with a cache line size of 32 bytes, only 0.23% of the remote references suffer from contention, the average remote latency is 82 cycles, and the running time is only 4.0 M cycles. (By way of comparison, Gaussian elimination under row-major allocation takes 30.9 M cycles on 200 processors when the cache line size is 32 bytes.) If we both double the memory latency and reduce the cache line size to 32 bytes, then only 0.9% of the remote references suffer from contention, the average remote latency rises slightly to 98 cycles (where the minimum is now 92 cycles), and the running time increases to 4.7 M cycles. Thus, the enormous performance advantages of block-column allocation are relatively insensitive to memory latency and cache line size.

The conclusion that block-column allocation can effectively eliminate the effects of contention holds even if we allocate multiple data rows to a memory module (rather than assign each row of the matrix to a separate page, and treat each page as a memory module). As long as consecutive rows are allocated in different memory modules, there is no significant contention for data within a memory module other than the contention measured in our simulations.

As a final observation, we note that Gaussian elimination runs slightly faster on 50 processors under row-major allocation than under block-column allocation. In this case, the additional addressing costs of block-column allocation outweigh the benefits associated with reducing memory contention. We will examine those costs in the next section.

Application	Running Time		
	50 processors	100 processors	200 processors
Gaussian elimination	7.7	4.5	2.96
Matrix inversion	25.3	15.3	10.1
Transitive closure	21.3	11.8	6.4
All pairs	15.4	10.5	10.3

Table 6: Effect of memory contention on running time (in millions of cycles) under block-column allocation.

Application	Running Time			
	Row-major (no contention)	Row-major (contention)	Block-column (no contention)	Block-column (contention)
Gaussian elimination	2.4	15.6	2.7	3.0
Matrix inversion	7.7	26.0	8.7	10.1
Transitive closure	6.1	9.0	6.3	6.4
All pairs	4.0	136.8	4.4	10.3

Table 7: Running time (in millions of cycles) with and without memory contention on 200 processors.

3.2 Overhead in Block-Column Allocation

As we discussed earlier, block-column allocation can be viewed as two loop transformations: strip-mining followed by loop interchange. The effect of strip-mining is to replace one loop with two, thereby increasing loop overhead. This overhead is not present when using row-major allocation, and therefore increases the running time of any program using block-column allocation, unless offset by a reduction in memory contention.

Table 7 illustrates the tradeoff between the overhead associated with block-column allocation and the memory contention associated with row-major allocation. In the absence of memory contention (that is, under the assumption that a memory module can satisfy any number of requests simultaneously), all of our programs execute 3-15% faster on 200 processors using row-major allocation, due to the overhead associated with block-column allocation. When memory contention is included, block-column allocation clearly dominates, improving performance by an order of magnitude in the case of all-pairs shortest paths. Recall from Tables 3 and 6 that block-column allocation performs significantly better on 50 processors only for those programs with a large amount of contention (matrix inversion and all-pairs shortest paths). For programs with lower contention levels, block-column allocation performs either slightly better (transitive closure) or slightly worse (Gaussian elimination) than row-major allocation on 50 processors. These data suggest that it is not always obvious how to resolve the tradeoffs involved. In the next section we analyze these tradeoffs to determine the circumstances under which to use block-column allocation.

4 Determining When to Use Block-Column Allocation

The previous section presented examples of the benefits of block-column allocation, and mentioned some of the tradeoffs associated with the technique. This section develops analytical models that explain why block-column allocation usually outperforms row-major allocation, and under what circumstances block-column allocation outperforms logarithmic broadcasting.

In each case, it's necessary to consider the two kinds of producer-consumer synchronization separately: barrier synchronization and lock synchronization. Under barrier synchronization, we assume that each task begins trying to access a new matrix row immediately after the barrier. This leads to a different analysis from lock synchronization, in which tasks access rows after a lock is set. Under lock synchronization, conflicts in accessing a matrix row are less frequent.

We first analyze block-column allocation and row-major allocation, and show that under each synchronization scheme, there exists some number of processors beyond which block-column allocation is always preferable to row-major allocation. Then, we analyze logarithmic broadcasting and show that making the proper choice between blocked-column allocation and logarithmic broadcasting depends both on the number of processors used to solve the problem, and on the type of synchronization used in the program.

The metric we will use in our comparison is the increase in running time over the optimal case, which has no memory contention and no additional instruction overhead. We measure the running time of the optimal case by simulating the simplest program (row-major allocation) on a system with infinite memory bandwidth (but nonzero memory latency).

Our purpose in performing these analyses is not to develop highly detailed models that can be used to predict the performance of programs. We focus instead on simple models that provide insight into reasons for preferring one technique over another, and that serve as a means of verifying our understanding of the tradeoffs involved.

4.1 Modeling the Costs of Block-Column and Row-Major Allocation

In our example applications, row-major allocation admits a simple loop structure, but suffers from memory contention. Block-column allocation alleviates memory contention, but introduces loop overhead, which results from strip mining over fairly small strips (i.e., the size of a cache line). To compare these two techniques, we must compare the relative impact of contention and strip mining overhead as we scale the number of processors.

For a given cache line size and matrix size, the loop overhead introduced by strip mining is a constant number of cycles. These cycles are distributed among the various processors, and therefore have a decreasing effect on running time as we increase the number of processors. The contention effects under block-column allocation depend on the form of synchronization. If processes are loosely synchronized (as is the case when we use locks), then the overhead introduced by block-column allocation is almost entirely attributed to loop overhead as follows:

$$BCA(P) = \frac{L}{P} + K_1$$

where L is the execution time of the additional instructions introduced by strip mining, and P is the number of processors (assuming good load balance). K_1, which is typically small relative to

Application	Running Time		
	50 processors	100 processors	200 processors
Optimal Gauss	6.5	3.7	2.4
Block-Column Gauss (Locks)	7.7	4.5	3.0
Optimal All pairs	12.4	6.7	4.0
Block-Column All pairs (Barriers)	15.4	10.5	10.3

Table 8: The running time of Gauss and All pairs (in millions of cycles) under block-column allocation, compared to optimal.

L, represents the small amount of contention that still occurs under lock synchronization. We find that the quantity K_1 is fixed for each of our programs.

Block-column allocation can suffer from memory contention when using barrier synchronization, but only for the first cache line of a row. Subsequent accesses to the same row are skewed by the serial access to the first cache line. The overhead of block-column allocation in this case is:

$$BCA(P) = \frac{L}{P} + RTP$$

where R is the number of rows in the matrix, and T is the transfer time of a cache line (82 cycles).

As seen in Table 8, our experimental results agree with this analysis. For Gaussian elimination, we measure L as approximately 50M cycles and K_1 as approximately 300,000 cycles. For all-pairs shortest paths, we measure L as approximately 70M cycles; from the program, we know that R is 400, and as noted above, $T = 82$. These parameters result in good agreement with the data in all cases.

In contrast, row-major allocation adds no additional loop overhead. However, it suffers serious contention under both barrier and lock synchronization. Under barrier synchronization, all processors contend for the entire row. Since all rows are eventually required by all processors, row-major allocation under barrier synchronization adds overhead equal to the cost of transferring the entire matrix, times P. This is because the last processor to receive a row will get it after $P-1$ other row transfers have completed. Under barrier synchronization, all the other processors will be forced to wait for the last processor at the next barrier, so all are slowed equally. In other words:

$$RMA(P) = \frac{M}{E}TP$$

where M is the number of elements in the entire matrix, and E is the number of elements per cache line.

Under lock synchronization, contention occurs due to random conflicts between processors, as before. However, random conflicts are more common, since processors access a single module repeatedly while transferring a row, and the demand for a particular row tends to be greatest immediately after it is produced. In fact, we can determine from the characteristics of our simulated machine that under row-major allocation, it only requires 8 processors transferring rows to saturate a memory module. Since the network trip lasts for 72 cycles, but the memory access itself only

Application	Running Time		
	50 processors	100 processors	200 processors
Optimal Gauss	6.5	3.7	2.4
Row-Major Gauss (Locks)	7.4	8.5	15.6
Optimal All pairs	12.4	6.7	4.0
Row-Major All pairs (Barriers)	43.0	71.3	136.8

Table 9: The running time of Gauss and All pairs (in millions of cycles) under row-major allocation, compared to optimal.

takes 10 cycles (which we will call *service time*), no more than 7 consecutive memory accesses can occur during a network trip.

Beyond a certain number of processors, we can expect that at any point in time, at least one memory module is saturated. This observation holds because there are only a fixed number of memories in use; adding more processors adds to the number of requests sent to each memory. The delay caused by a memory module's saturation is eventually propagated to all processes, since each processor (in addition to consuming rows) is producing a row that eventually the other processors will need.

Thus, although it is difficult to model the random contention for memory when the number of processors is small, we can provide an estimate of overhead when the number of processors is large. This estimate is based on the assumption that at any point in time, some module is saturated. We can then see that each additional processor adds an additional service time to the transfer of each cache line, since the additional processor will likely access the module while it is saturated. This means that each additional processor adds the cost of an entire matrix's memory service time, or 10 cycles times the number of cache lines in an entire matrix. So we estimate the overhead of row-major allocation, for large P, and lock synchronization, as:

$$RMA(P) = \frac{M}{E} C(P - \theta)$$

where C is the memory's service time (10 cycles), and θ is the threshold number of processors beyond which the system shows memory saturation.

As seen in Table 9, our experimental results for row-major allocation generally confirm our analysis. For all-pairs shortest paths, where $M = 400^2$, our predictions are about 30% too high; however, these running times are extremely long and our model predicts them well enough for comparison purposes with block-column allocation. For Gaussian elimination, we determine by inspecting the data that memory saturation is reached at about 40 processors, so $\theta = 40$; also, since pivot rows only constitute the upper half of the matrix in Gaussian elimination, $M = 512^2/2$. Our model of overhead for lock synchronization is then quite accurate.

Using this analysis, we can determine when the extra cost of block-column allocation is worth paying in exchange for the reduction in contention that it provides. Figure 1 shows plots of the analytic models developed above, for the cases of all-pairs shortest paths (on the left) and Gaussian elimination (on the right). The all-pairs graph shows that under the high contention costs of barrier

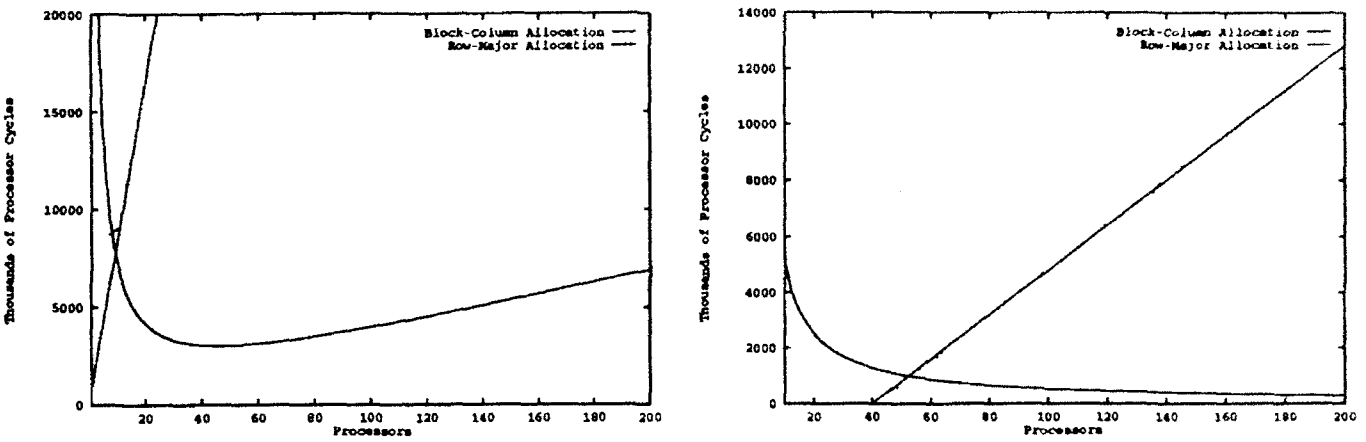


Figure 1: Overhead of Row Major Allocation compared to Block-Column Allocation for Barrier Synchronization (Left) and Lock Synchronization (Right)

synchronization, block-column allocation is preferable even on as few as 10 processors. Beyond about 50 processors, the cost of block-column allocation begins to rise, but at a slower rate than the cost of row-major allocation. This trend reflects the difference between contending for the first cache line of the row in the block-column case, and contending for the entire row in the row-major case.

The analytic models for lock synchronization in Gaussian elimination are plotted on the right side of Figure 1. Since contention under lock synchronization starts more slowly than under barriers, more processors are required before block-column allocation is preferred over row-major, but the same basic effect is observed: beyond some number of processors (in this case about 50) block-column allocation is always preferable.

4.2 Comparing Block-Column Allocation and Logarithmic Broadcasting

The previous section showed that as the number of processors increases, eventually there comes a point when it is more profitable to use block-column allocation over row-major allocation. However, to adequately assess when to use block-column allocation, we must compare it to the best known alternative: logarithmic broadcasting.

We implemented two versions of broadcasting for the row-major Gaussian elimination program. One version is consumer-driven: the producer sets a flag indicating when data is ready, and the consumers copy the data; the other is producer-driven: the producer copies the data for the consumers. In the producer-driven implementation, the copies must occur in sequence. In the consumer-driven implementation, multiple consumers can overlap time spent in the network, so many copy operations can proceed in parallel. Thus, the consumer-driven technique performs significantly better, which is why we only present results for that technique.

As pointed out in the last section, 8 processors reading a row can saturate a memory module when the memory latency is 10 cycles and the network latency is 72 cycles; however, as long as

Application	Running Time		
	50 processors	100 processors	200 processors
Optimal Gauss	6.5	3.7	2.4
Log. Broadcasting Gauss (Locks)	7.4	4.7	3.4
Optimal All pairs	12.4	6.7	4.0
Log. Broadcasting All pairs (Barriers)	15.9	10.3	7.8

Table 10: The running time of Gauss and All pairs (in millions of cycles) under logarithmic broadcasting, compared to optimal.

the number of processors contending is less than 8, each processor is delayed only a small amount. Thus, in our simulated machine, logarithmic broadcasting should not use a tree of degree greater than 8. With this assumption, logarithmic broadcasting can completely eliminate contention when used with lock synchronization. This is because the condition in which some memory module is always saturated does not occur, as it did under simple row-major allocation. Memory modules do not saturate since the complete broadcast of each row is implemented using a much larger set of memory modules, and the number of processors accessing a single module will never be greater than the degree of the tree.

For this reason we can estimate the cost of logarithmic broadcasting under lock synchronization as a constant, which is equal to the extra instructions and synchronization necessary to implement the technique. Thus,

$$LB(P) = K_2$$

where K_2 depends on the specific program. Interestingly, in the programs we studied, K_2 was significant; for example, in Gaussian elimination, K_2 = 1.0M cycles. This occurs partly due to the synchronization needed to access broadcast buffers. Ideally each row would have a broadcast buffer on each processor, but that would require expanding the memory usage of the program by a factor of P, which is impractical. Since the amount of buffer space used for row broadcast on each processor must be bounded, buffer space must be re-used, which requires synchronization.

In contrast, under barrier synchronization, the cost of logarithmic broadcasting is not independent of P. The broadcast of each row requires d steps, where $d+1$ is the depth of the broadcast tree.[3] For a tree of degree r, each step requires r row transfers. The first row causes a delay equal to its transfer time; the other rows cause a delay equal only to their memory service times (as discussed earlier in this section). Thus we can estimate the overhead of logarithmic broadcasting under barrier synchronization as:

$$LB(P) = d\frac{M}{E}T + d(r-1)\frac{M}{E}C$$

where d is proportional to $\lceil \log_r P \rceil$.

In our experiments we held d equal to 3, while we varied r to attain the lowest possible value consistent with $d = 3$. For the 50 processor case, we set $r = 4$; for $P = 100$, $r = 5$; and for

[3]For a tree of degree r, the depth of the broadcast tree is roughly $\lceil \log_r P \rceil$, although details of how the tree is constructed can change this value by 1 in some cases. In all our experiments, $d = 3$.

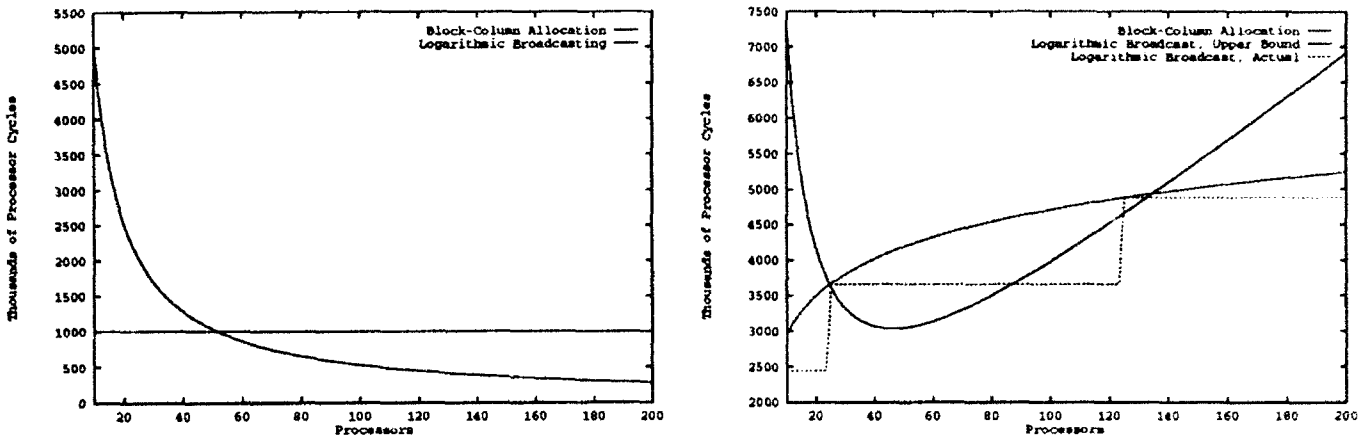


Figure 2: Overhead of Logarithmic Broadcasting compared to Block-Column Allocation for Lock Synchronization (Left) and Barrier Synchronization (Right)

$P = 200$, $r = 6$. Table 10 shows the results of our experiments with All pairs and Gaussian elimination under logarithmic broadcasting, and compares them to their ideal cases. The table shows that $K_2 = 1.0$M cycles is a good estimate of the constant overhead for Gaussian elimination under logarithmic broadcasting. It also shows that our estimate of the overhead due to logarithmic broadcasting under barriers in all-pairs shortest paths is fairly accurate.

Figure 2 shows how the two techniques compare. The comparison for lock synchronization is on the left, while the comparison for barrier synchronization is on the right. For lock synchronization, beyond about 50 processors, block-column allocation performs better than logarithmic broadcasting. This is because the fixed overhead under block-column allocation is lower than that under logarithmic broadcasting. Since contention is much less severe under lock synchronization, the extra cycles required to implement logarithmic broadcasting are more expensive than necessary; block-column allocation is preferable due to its simplicity.

The situation is different for barrier synchronization, as shown on the right side of Figure 2. This figure shows the overhead of block-column allocation compared to logarithmic broadcasting using a tree of fixed degree (equal to 5). The step-function nature of the logarithmic broadcasting curve is due to changes in the depth of the tree as the number of processors increases. The figure also shows an upper bound on logarithmic broadcasting to show that as P grows large, logarithmic broadcasting eventually outperforms block-column allocation everywhere. This figure shows that under barrier synchronization contention is so severe that the linearly increasing costs of accessing the first cache line in each row under block-column allocation eventually grow larger than the logarithmically increasing costs of broadcast.

Figure 2 shows that for large numbers of processors, logarithmic broadcasting is best when using barrier synchronization, but block-column allocation is best when using lock synchronization. It also shows that for small numbers of processors, the situation is reversed: block-column allocation is best when using barrier synchronization, while logarithmic broadcasting is best when using lock synchronization.

5 Conclusions

In this paper we used detailed simulations of application kernels to show that memory contention can substantially degrade the performance of SPMD computations on large-scale shared-memory multiprocessors. We showed that under row-major allocation, memory contention is due to synchronized access to entire rows of a matrix, rather than simultaneous accesses to isolated data elements. We also showed that block-column allocation, which divides the rows of a matrix into cache lines, and distributes the cache lines containing each row among multiple memory modules, dramatically reduces memory contention, and therefore performs much better than row-major allocation on large-scale machines.

We analyzed the costs associated with block-column allocation and logarithmic broadcasting, and showed how the choice between these two techniques for alleviating memory contention depends both on the type of synchronization used and the number of processors. For large numbers of processors, logarithmic broadcasting is best when using barrier synchronization, but block-column allocation is best when using lock synchronization. For small numbers of processors, the situation is reversed: block-column allocation is best when using barrier synchronization, while logarithmic broadcasting is best when using lock synchronization. Since the use of barrier synchronization exacerbates memory contention, we conclude that block-column allocation and lock-based synchronization is the most effective combination for reducing memory contention in SPMD matrix computations on large-scale machines.

References

[Agarwal *et al.*, 1992] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum, "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor," In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[Amestoy *et al.*, 1992] P. R. Amestoy, M. J. Dayde, I. S. Duff, and P. Morere, "Linear Algebra Computations on the BBN TC2000," In *Parallel Processing: CONPAR 92 - VAPP V*, pages 319–330, Lyon, France, September 1992.

[BBN, 1989] BBN Advanced Computers Inc., *Inside the TC2000*, 1989.

[Bianchini *et al.*, 1993] R. Bianchini, M. E. Crovella, L. Kontothanasis, and T. J. LeBlanc, "Hot Spot Removal in Scalable Cache-Coherent Multiprocessors," Technical Report 448, Department of Computer Science, University of Rochester, April 1993.

[Brooks and Warren, 1991] E. D. Brooks and K. H. Warren, "The 1991 MPCI Yearly Report: The Attack of the Killer Micros," Technical report, Lawrence Livermore National Laboratory, 1991.

[Davis *et al.*, 1991] Helen Davis, Stephen R. Goldschmidt, and John Hennessy, "Multiprocessor Simulation and Tracing Using Tango," In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II–99 – II–107, August 1991.

[Geist *et al.*, 1987] G. A. Geist, M. T. Heath, and E. NG, "Parallel Algorithms for Matrix Computations," In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *Characteristics of Parallel Algorithms*. MIT Press, 1987.

[Glenn *et al.*, 1991] R. R. Glenn, D. V. Pryor, J. M. Conroy, and T. Johnson, "Characterizing Memory Hot Spots in a Shared-Memory MIMD Machine," In *Proceeedings of Supercomputing '91*, pages 554–566, Albuquerque, NM, November 1991.

[Gottlieb *et al.*, 1983] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[Lenoski *et al.*, 1992] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance," In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.

[Mellor-Crummey and Scott, 1991] J. M. Mellor-Crummey and M. L. Scott, "Synchronization Without Contention," In *Proceedings of the Fourth International Conference Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Santa Clara, CA, 8-11 April 1991.

[Ortega and Romine, 1988] J. M. Ortega and C. H. Romine, "The ijk Forms of Factorization Methods II. Parallel Systems," *Parallel Computing*, 7:149–162, 1988.

[Pfister *et al.*, 1985] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[Pfister and Norton, 1985] G. F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[Saad and Schultz, 1989] Y. Saad and M. H. Schultz, "Data Communication in Parallel Architectures," *Parallel Computing*, 11(2):131–150, August 1989.

[Wittie and Maples, 1989] Larry Wittie and Creve Maples, "MERLIN: Massively Parallel Heterogeneous Computing," In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–142 – I–150, 1989.

[Yew *et al.*, 1987] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.